Chapter 4

Methods and Encapsulation

OCA EXAM OBJECTIVES COVERED IN THIS CHAPTER:

✓ **Working with Methods and Encapsulation**

- Create methods with arguments and return values; including overloaded methods
- Apply the static keyword to methods and fields
- Create and overload constructors; include impact on default constructors
- Apply access modifiers
- Apply encapsulation principles to a class
- Determine the effect upon object references and primitive values when they are passed into methods that change the values

✓ **Working with Selected classes from the Java API**

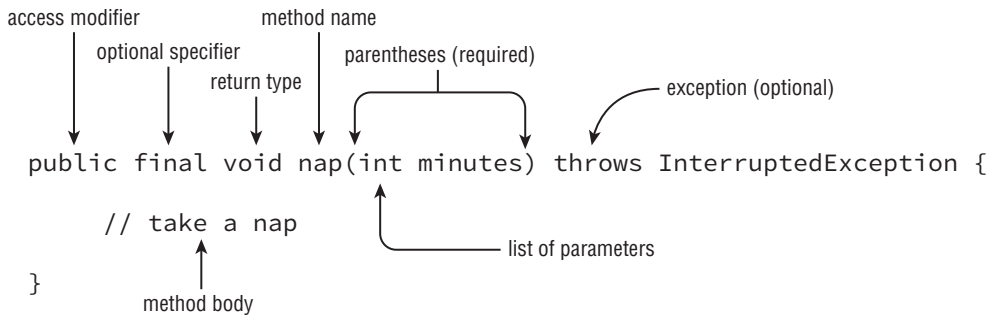
- Write a simple Lambda expression that consumes a Lambda Predicate expression

In previous chapters, we've used methods and constructors without examining them in detail. In this chapter, we'll explore methods and constructors in depth and cover everything you need to know about them for the OCA exam. (Well, almost—Chapter 5, "Class Design," will explain the effects of inheritance on both methods and constructors.) This chapter discusses instance variables, the `final` keyword, access modifiers, and initialization. You'll also learn how to write a simple lambda expression.

Designing Methods

Every interesting Java program we've seen has had a `main()` method. We can write other methods, too. For example, we can write a basic method to take a nap, as shown in Figure 4.1.

FIGURE 4.1 Method signature



This is called a *method declaration*, which specifies all the information needed to call the method. There are a lot of parts, and we'll cover each one in more detail. Table 4.1 is a brief reference to the elements of a method declaration. Don't worry if it seems like a lot of information—by the time you finish this chapter, it will all fit together.

TABLE 4.1 Parts of a method declaration

Element	Value in `nap()` example	Required?
Access modifier	`public`	No
Optional specifier	`final`	No

Element	Value in nap() example	Required?
Return type	`void`	Yes
Method name	`nap`	Yes
Parameter list	`(int minutes)`	Yes, but can be empty parentheses
Optional exception list	`throws InterruptedException`	No
Method body	`{ // take a nap }`	Yes, but can be empty braces

To call this method, just type its name, followed by a single `int` value in parentheses:

```
nap(10);
```

Let's start by taking a look at each of these parts of a basic method.

Access Modifiers

Java offers four choices of access modifier:

public The method can be called from any class.

private The method can only be called from within the same class.

protected The method can only be called from classes in the same package or subclasses. You'll learn about subclasses in Chapter 5.

Default (Package Private) Access The method can only be called from classes in the same package. This one is tricky because there is no keyword for default access. You simply omit the access modifier.

There's a `default` keyword in Java. You saw it in the `switch` statement in Chapter 2, "Operators and Statements," and you'll see it again in the next chapter when we discuss interfaces. It's not used for access control.

We'll explore the impact of the various access modifiers later in this chapter. For now, just master identifying valid syntax of methods. The exam creators like to trick you by putting method elements in the wrong order or using incorrect values.

We'll have practice examples as we go through each of the method elements in this section. Make sure you understand why each of these is a valid or invalid method declaration. Pay attention to the access modifiers as you figure out what is wrong with the ones that don't compile:

```
public void walk1() {}
default void walk2() {} // DOES NOT COMPILE
```

```
void public walk3() {} // DOES NOT COMPILE
void walk4() {}
```

`walk1()` is a valid method declaration with public access. `walk4()` is a valid method declaration with default access. `walk2()` doesn't compile because default is not a valid access modifier. `walk3()` doesn't compile because the access modifier is specified after the return type.

Optional Specifiers

There are a number of optional specifiers, but most of them aren't on the exam. Optional specifiers come from the following list. Unlike with access modifiers, you can have multiple specifiers in the same method (although not all combinations are legal). When this happens, you can specify them in any order. And since it is optional, you can't have any of them at all. This means you can have zero or more specifiers in a method declaration.

static Covered later in this chapter. Used for class methods.

abstract Covered in Chapter 5. Used when not providing a method body.

final Covered in Chapter 5. Used when a method is not allowed to be overridden by a subclass.

synchronized On the OCP but not the OCA exam.

native Not on the OCA or OCP exam. Used when interacting with code written in another language such as C++.

strictfp Not on the OCA or OCP exam. Used for making floating-point calculations portable.

Again, just focus on syntax for now. Do you see why these compile or don't compile?

```
public void walk1() {}
public final void walk2() {}
public static final void walk3() {}
public final static void walk4() {}
public modifier void walk5() {} // DOES NOT COMPILE
public void final walk6() {} // DOES NOT COMPILE
final public void walk7() {}
```

`walk1()` is a valid method declaration with no optional specifier. This is okay; it is optional, after all. `walk2()` is a valid method declaration, with `final` as the optional specifier. `walk3()` and `walk4()` are valid method declarations with both `final` and `static` as optional specifiers. The order of these two keywords doesn't matter. `walk5()` doesn't

compile because *modifier* is not a valid optional specifier. `walk6()` doesn't compile because the optional specifier is after the return type.

`walk7()` does compile. Java allows the optional specifiers to appear before the access modifier. This is a weird case and not one you need to know for the exam. We are mentioning it so you don't get confused when practicing.

Return Type

The next item in a method declaration is the return type. The return type might be an actual Java type such as `String` or `int`. If there is no return type, the `void` keyword is used. This special return type comes from the English language: *void* means without contents. In Java, we have no type there.

Remember that a method must have a return type. If no value is returned, the return type is `void`. You cannot omit the return type.

When checking return types, you also have to look inside the method body. Methods with a return type other than `void` are required to have a `return` statement inside the method body. This return statement must include the primitive or object to be returned. Methods that have a return type of `void` are permitted to have a `return` statement with no value returned or omit the `return` statement entirely.

Ready for some examples? Can you explain why these methods compile or don't?

```
public void walk1() { }
public void walk2() { return; }
public String walk3() { return ""; }
public String walk4() { }  // DOES NOT COMPILE
public walk5() { }  // DOES NOT COMPILE
String walk6(int a) { if (a == 4) return ""; }  // DOES NOT COMPILE
```

Since the return type of `walk1()` is `void`, the return statement is optional. `walk2()` shows the optional return statement that correctly doesn't return anything. `walk3()` is a valid method with a `String` return type and a return statement that returns a `String`. `walk4()` doesn't compile because the return statement is missing. `walk5()` doesn't compile because the return type is missing.

`walk6()` is a little tricky. There is a return statement, but it doesn't always get run. If `a` is 6, the return statement doesn't get executed. Since the `String` always needs to be returned, the compiler complains.

When returning a value, it needs to be assignable to the return type. Imagine there is a local variable of that type to which it is assigned before being returned. Can you think of how to add a line of code with a local variable in these two methods?

```
int integer() {
  return 9;
}
int long() {]
  return 9L; // DOES NOT COMPILE
}
```

It is a fairly mechanical exercise. You just add a line with a local variable. The type of the local variable matches the return type of the method. Then you return that local variable instead of the value directly:

```
int integerExpanded() {
  int temp = 9;
  return temp;
}
int longExpanded() {
  int temp = 9L; // DOES NOT COMPILE
  return temp;
}
```

This shows more clearly why you can't return a `long` primitive in a method that returns an `int`. You can't stuff that `long` into an `int` variable, so you can't return it directly either.

Method Name

Method names follow the same rules as we practiced with variable names in Chapter 1, "Java Building Blocks." To review, an identifier may only contain letters, numbers, $, or _. Also, the first character is not allowed to be a number, and reserved words are not allowed. By convention, methods begin with a lowercase letter but are not required to. Since this is a review of Chapter 1, we can jump right into practicing with some examples:

```
public void walk1() { }
public void 2walk() { } // DOES NOT COMPILE
public walk3 void() { } // DOES NOT COMPILE
public void Walk_$() { }
public void() { } // DOES NOT COMPILE
```

`walk1()` is a valid method declaration with a traditional name. `2walk()` doesn't compile because identifiers are not allowed to begin with numbers. `walk3()` doesn't compile because the method name is before the return type. `Walk_$()` is a valid method declaration. While it certainly isn't good practice to start a method name with a capital letter and end with punctuation, it is legal. The final line of code doesn't compile because the method name is missing.

Parameter List

Although the parameter list is required, it doesn't have to contain any parameters. This means you can just have an empty pair of parentheses after the method name, such as `void nap(){}`. If you do have multiple parameters, you separate them with a comma. There are a couple more rules for the parameter list that you'll see when we cover varargs shortly. For now, let's practice looking at method signatures with "regular" parameters:

```
public void walk1() { }
public void walk2 { } // DOES NOT COMPILE
public void walk3(int a) { }
public void walk4(int a; int b) { }  // DOES NOT COMPILE
public void walk5(int a, int b) { }
```

`walk1()` is a valid method declaration without any parameters. `walk2()` doesn't compile because it is missing the parentheses around the parameter list. `walk3()` is a valid method declaration with one parameter. `walk4()` doesn't compile because the parameters are separated by a semicolon rather than a comma. Semicolons are for separating statements, not parameter lists. `walk5()` is a valid method declaration with two parameters.

Optional Exception List

In Java, code can indicate that something went wrong by throwing an exception. We'll cover this in Chapter 6, "Exceptions." For now, you just need to know that it is optional and where in the method signature it goes if present. In the example, `InterruptedException` is a type of `Exception`. You can list as many types of exceptions as you want in this clause separated by commas. For example:

```
public void zeroExceptions() { }
public void oneException() throws IllegalArgumentException { }
public void twoExceptions() throws
  IllegalArgumentException, InterruptedException { }
```

You might be wondering what methods do with these exceptions. The calling method can throw the same exceptions or handle them. You'll learn more about this in Chapter 6.

Method Body

The final part of a method declaration is the method body (except for abstract methods and interfaces, but you don't need to know about either of those until next chapter). A method body is simply a code block. It has braces that contain zero or more Java statements. We've spent several chapters looking at Java statements by now, so you should find it easy to figure out why these compile or don't:

```
public void walk1() { }
public void walk2; // DOES NOT COMPILE
public void walk3(int a) { int name = 5; }
```

`walk1()` is a valid method declaration without an empty method body. `walk2()` doesn't compile because it is missing the braces around the empty method body. `walk3()` is a valid method declaration with one statement in the method body.

You've made it through the basics of identifying correct and incorrect method declarations. Now we can delve into more detail.

Working with Varargs

As you saw in the previous chapter, a method may use a vararg parameter (variable argument) as if it is an array. It is a little different than an array, though. A vararg parameter must be the last element in a method's parameter list. This implies you are only allowed to have one vararg parameter per method.

Can you identify why each of these does or doesn't compile? (Yes, there is a lot of practice in this chapter. You have to be really good at identifying valid and invalid methods for the exam.)

```
public void walk1(int... nums) { }
public void walk2(int start, int... nums) { }
public void walk3(int... nums, int start) { } // DOES NOT COMPILE
public void walk4(int... start, int... nums) { } // DOES NOT COMPILE
```

`walk1()` is a valid method declaration with one vararg parameter. `walk2()` is a valid method declaration with one `int` parameter and one vararg parameter. `walk3()` and `walk4()` do not compile because they have a vararg parameter in a position that is not the last one.

When calling a method with a vararg parameter, you have a choice. You can pass in an array, or you can list the elements of the array and let Java create it for you. You can even omit the vararg values in the method call and Java will create an array of length zero for you.

Finally! We get to do something other than identify whether method declarations are valid. Instead we get to look at method calls. Can you figure out why each method call outputs what it does?

```
15: public static void walk(int start, int... nums) {
16:  System.out.println(nums.length);
17: }
18: public static void main(String[] args) {
19:  walk(1);                    // 0
20:  walk(1, 2);                 // 1
```

```
21:  walk(1, 2, 3);                 // 2
22:  walk(1, new int[] {4, 5});     // 2
23: }
```

Line 19 passes 1 as `start` but nothing else. This means Java creates an array of length 0 for `nums`. Line 20 passes 1 as `start` and one more value. Java converts this one value to an array of length 1. Line 21 passes 1 as `start` and two more values. Java converts these two values to an array of length 2. Line 22 passes 1 as `start` and an array of length 2 directly as `nums`.

You've seen that Java will create an empty array if no parameters are passed for a vararg. However, it is still possible to pass `null` explicitly:

```
walk(1, null);     // throws a NullPointerException
```

Since `null` isn't an `int`, Java treats it as an array reference that happens to be `null`. It just passes on the `null` array object to `walk`. Then the `walk()` method throws an exception because it tries to determine the length of `null`.

Accessing a vararg parameter is also just like accessing an array. It uses array indexing. For example:

```
16: public static void run(int... nums) {
17:   System.out.println(nums[1]);
18: }
19: public static void main(String[] args) {
20:   run(11, 22);     // 22
21: }
```

Line 20 calls a vararg parameter two parameters. When the method gets called, it sees an array of size 2. Since indexes are 0 based, 22 is printed.

Applying Access Modifiers

You already saw that there are four access modifiers: `public`, `private`, `protected`, and default access. We are going to discuss them in order from most restrictive to least restrictive:

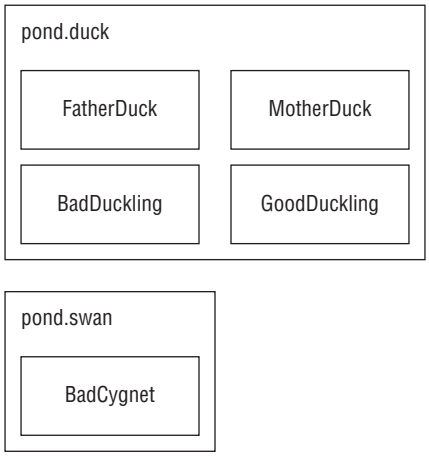
- `private:` Only accessible within the same class
- default (package private) access: `private` and other classes in the same package
- `protected:` default access and child classes
- `public:` `protected` and classes in the other packages

Private Access

Private access is easy. Only code in the same class can call private methods or access private fields.

Before we start, take a look at Figure 4.2. It shows the classes we'll use to explore private and default access. The big boxes are the names of the packages. The smaller boxes inside them are the classes in each package. You can refer back to this figure if you want to quickly see how the classes relate.

FIGURE 4.2 Classes used to show private and default access



This is perfectly legal code because everything is one class:

```
1: package pond.duck;
2: public class FatherDuck {
3:  private String noise = "quack";
4:  private void quack() {
5:    System.out.println(noise);     // private access is ok
6:  }
7:  private void makeNoise() {
8:    quack();                       // private access is ok
9:  } }
```

So far, so good. `FatherDuck` makes a call to private method `quack()` on line 8 and uses private instance variable *`noise`* on line 5.

Now we add another class:

```
1: package pond.duck;
2: public class BadDuckling {
3:  public void makeNoise() {
4:    FatherDuck duck = new FatherDuck();
5:    duck.quack();                        // DOES NOT COMPILE
6:    System.out.println(duck.noise);      // DOES NOT COMPILE
7:  } }
```

`BadDuckling` is trying to access members it has no business touching. On line 5, it tries to access a private method in another class. On line 6, it tries to access a private instance variable in another class. Both generate compiler errors. Bad duckling!

Our bad duckling is only a few days old and doesn't know better yet. Luckily, you know that accessing private members of other classes is not allowed and you need to use a different type of access.

Default (Package Private) Access

Luckily, `MotherDuck` is more accommodating about what her ducklings can do. She allows classes in the same package to access her members. When there is no access modifier, Java uses the default, which is package private access. This means that the member is "private" to classes in the same package. In other words, only classes in the package may access it.

```
package pond.duck;

public class MotherDuck {
  String noise = "quack";
  void quack() {
    System.out.println(noise);     // default access is ok
  }
  private void makeNoise() {
    quack();                    // default access is ok
  } }
```

`MotherDuck` can call `quack()` and refer to *`noise`*. After all, members in the same class are certainly in the same package. The big difference is `MotherDuck` lets other classes in the same package access members (due to being package private) whereas `FatherDuck` doesn't (due to being private). `GoodDuckling` has a much better experience than `BadDuckling`:

```
package pond.duck;

public class GoodDuckling {
  public void makeNoise() {
    MotherDuck duck = new MotherDuck();
    duck.quack();                          // default access
    System.out.println(duck.noise);           // default access
  } }
```

`GoodDuckling` succeeds in learning to `quack()` and make *`noise`* by copying its mother. Notice that all the classes we've covered so far are in the same package `pond.duck`. This allows default (package private) access to work.

In this same pond, a swan just gave birth to a baby swan. A baby swan is called a cygnet. The cygnet sees the ducklings learning to quack and decides to learn from `MotherDuck` as well.

```
package pond.swan;
import pond.duck.MotherDuck;          // import another package
```

```
public class BadCygnet {
  public void makeNoise() {
    MotherDuck duck = new MotherDuck();
    duck.quack();                        // DOES NOT COMPILE
    System.out.println(duck.noise);      // DOES NOT COMPILE
  } }
```

Oh no! `MotherDuck` only allows lessons to other ducks by restricting access to the `pond.duck` package. Poor little `BadCygnet` is in the `pond.swan` package and the code doesn't compile.

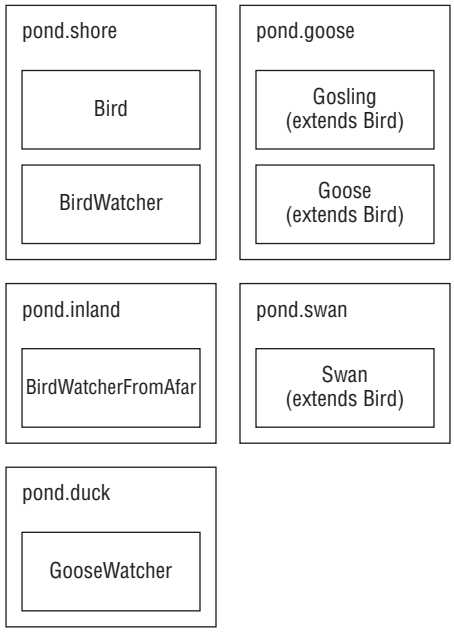
Remember that when there is no access modifier, only classes in the same package can access it.

Protected Access

Protected access allows everything that default (package private) access allows and more. The `protected` access modifier adds the ability to access members of a parent class. We'll cover creating subclasses in depth in Chapter 5. For now, we'll cover the simplest possible use of a child class.

Figure 4.3 shows the many classes we'll create in this section. There are a number of classes and packages, so don't worry about keeping them all in your head. Just check back with this figure as you go.

FIGURE 4.3 Classes used to show protected access



First, we create a `Bird` class and give `protected` access to its members:

```
package pond.shore;
public class Bird {
  protected String text = "floating";          // protected access
  protected void floatInWater() {               // protected access
    System.out.println(text);
  } }
```

Next we create a subclass:

```
package pond.goose;
import pond.shore.Bird;                 // in a different package
public class Gosling extends Bird {     // extends means create subclass
  public void swim() {
    floatInWater();                 // calling protected member
    System.out.println(text);       // calling protected member
  } }
```

This is a very simple subclass. It `extends` the `Bird` class. Extending means creating a subclass that has access to any `protected` or `public` members of the parent class. Running this code prints `floating` twice: once from calling `floatInWater()` and once from the print statement in `swim()`. Since `Gosling` is a subclass of `Bird`, it can access these members even though it is in a different package.

Remember that `protected` also gives us access to everything that default access does. This means that a class in the same package as `Bird` can access its `protected` members.

```
package pond.shore;                     // same package as Bird
public class BirdWatcher {
  public void watchBird() {
    Bird bird = new Bird();
    bird.floatInWater();                // calling protected member
    System.out.println(bird.text);      // calling protected member
  } }
```

Since `Bird` and `BirdWatcher` are in the same package, `BirdWatcher` can access members of the *`bird`* variable. The definition of `protected` allows access to subclasses and classes in the same package. This example uses the same package part of that definition.

Now let's try the same thing from a different package:

```
package pond.inland;
import pond.shore.Bird;                 // different package than Bird
public class BirdWatcherFromAfar {
  public void watchBird() {
    Bird bird = new Bird();
    bird.floatInWater();                // DOES NOT COMPILE
```

```
    System.out.println(bird.text);     // DOES NOT COMPILE
  } }
```

`BirdWatcherFromAfar` is not in the same package as `Bird` and it doesn't inherit from `Bird`. This means that it is not allowed to access `protected` members of `Bird`.

Got that? Subclasses and classes in the same package are the only ones allowed to access `protected` members.

There is one gotcha for `protected` access. Consider this class:

```
1:  package pond.swan;
2:  import pond.shore.Bird;     // in different package than Bird
3:  public class Swan extends Bird {     // but subclass of bird
4:    public void swim() {
5:      floatInWater();               // package access to superclass
6:      System.out.println(text);     // package access to superclass
7:    }
8:    public void helpOtherSwanSwim() {
9:      Swan other = new Swan();
10:     other.floatInWater();          // package access to superclass
11:     System.out.println(other.text);// package access to superclass
12:   }
13:   public void helpOtherBirdSwim() {
14:     Bird other = new Bird();
15:     other.floatInWater();               // DOES NOT COMPILE
16:     System.out.println(other.text);      // DOES NOT COMPILE
17:   }
18: }
```

Take a deep breath. This is interesting. `Swan` is not in the same package as `Bird`, but does extend it—which implies it has access to the `protected` members of `Bird` since it is a subclass. And it does. Lines 5 and 6 refer to `protected` members via inheriting them.

Lines 10 and 11 also successfully use `protected` members of `Bird`. This is allowed because these lines refer to a `Swan` object. Swan inherits from `Bird` so this is okay. It is sort of a two-phase check. The `Swan` class is allowed to use `protected` members of `Bird` and we are referring to a `Swan` object. Granted, it is a `Swan` object created on line 9 rather than an inherited one, but it is still a `Swan` object.

Lines 15 and 16 do *not* compile. Wait a minute. They are almost exactly the same as lines 10 and 11! There's one key difference. This time a `Bird` reference is used. It is created on line 14. `Bird` is in a different package, and this code isn't inheriting from `Bird`, so it doesn't get to use `protected` members. Say what now? We just got through saying repeatedly that `Swan` inherits from `Bird`. And it does. However, the variable reference isn't a `Swan`. The code just happens to be in the `Swan` class.

It's okay to be confused. This is arguably one of the most confusing points on the exam. Looking at it a different way, the `protected` rules apply under two scenarios:

- A member is used without referring to a variable. This is the case on lines 5 and 6. In this case, we are taking advantage of inheritance and `protected` access is allowed.
- A member is used through a variable. This is the case on lines 10, 11, 15, and 16. In this case, the rules for the reference type of the variable are what matter. If it is a subclass, `protected` access is allowed. This works for references to the same class or a subclass.

We're going to try this again to make sure you understand what is going on. Can you figure out why these examples don't compile?

```
package pond.goose;
import pond.shore.Bird;
public class Goose extends Bird {
  public void helpGooseSwim() {
    Goose other = new Goose();
    other.floatInWater();
    System.out.println(other.text);
  }
  public void helpOtherGooseSwim() {
    Bird other = new Goose();
    other.floatInWater(); // DOES NOT COMPILE
    System.out.println(other.text); // DOES NOT COMPILE
  } }
```

The first method is fine. In fact, it is equivalent to the `Swan` example. `Goose` extends `Bird`. Since we are in the `Goose` subclass and referring to a `Goose` reference, it can access `protected` members. The second method is a problem. Although the object happens to be a `Goose`, it is stored in a `Bird` reference. We are not allowed to refer to members of the `Bird` class since we are not in the same package and `Bird` is not a subclass of `Bird`.

What about this one?

```
package pond.duck;
import pond.goose.Goose;
public class GooseWatcher {
  public void watch() {
    Goose goose = new Goose();
    goose.floatInWater();     // DOES NOT COMPILE
  } }
```

This code doesn't compile because we are not in the `Goose` class. The `floatInWater()` method is declared in `Bird`. `GooseWatcher` is not in the same package as `Bird`, nor does it

extend `Bird`. `Goose` extends `Bird`. That only lets `Goose` refer to `floatInWater()` and not callers of `Goose`.

If this is still puzzling, try it out. Type in the code and try to make it compile. Then reread this section. Don't worry—it wasn't obvious to the authors the first time either!

Public Access

Protected access was a tough concept. Luckily, the last type of access modifier is easy: `public` means anyone can access the member from anywhere.

```
package pond.duck;
public class DuckTeacher {
  public String name = "helpful";     // public access
  public void swim() {                // public access
    System.out.println("swim");
  } }
```

`DuckTeacher` allows access to any class that wants it. Now we can try it out:

```
package pond.goose;
import pond.duck.DuckTeacher;
public class LostDuckling {
  public void swim() {
    DuckTeacher teacher = new DuckTeacher();
    teacher.swim();                                  // allowed
    System.out.println("Thanks" + teacher.name);     // allowed
  } }
```

`LostDuckling` is able to refer to `swim()` and `name` on `DuckTeacher` because they are public. The story has a happy ending. `LostDuckling` has learned to swim and can find its parents—all because `DuckTeacher` made them public.

To review access modifiers, make sure you know why everything in Table 4.2 is true. Remember that a member is a method or field.

TABLE 4.2 Access modifiers

Can access	If that member is private?	If that member has default (package private) access?	If that member is protected?	If that member is public?
Member in the same class	Yes	Yes	Yes	Yes
Member in another class in same package	No	Yes	Yes	Yes

Can access	If that member is private?	If that member has default (package private) access?	If that member is protected?	If that member is public?
Member in a superclass in a different package	No	No	Yes	Yes
Method/field in a non-superclass class in a different package	No	No	No	Yes

Designing Static Methods and Fields

Except for the `main()` method, we've been looking at instance methods. Static methods don't require an instance of the class. They are shared among all users of the class. You can think of statics as being a member of the single class object that exist independently of any instances of that class.

Does Each Class Have Its Own Copy of the Code?

Each class has a copy of the instance variables. There is only one copy of the code for the instance methods. Each instance of the class can call it as many times as it would like. However, each call of an instance method (or any method) gets space on the stack for method parameters and local variables.

The same thing happens for static methods. There is one copy of the code. Parameters and local variables go on the stack.

Just remember that only data gets its "own copy." There is no need to duplicate copies of the code itself.

We have seen one static method since Chapter 1. The `main()` method is a static method. That means you can call it by the classname.

```
public class Koala {
  public static int count = 0;               // static variable
  public static void main(String[] args) {      // static method
    System.out.println(count);
  }
}
```

We said that the JVM basically calls `Koala.main()` to get the program started. You can do this too. We can have a `KoalaTester` that does nothing but call the `main()` method.

```
public class KoalaTester {
  public static void main(String[] args) {
    Koala.main(new String[0]);          // call static method
  }
}
```

Quite a complicated way to print 0, isn't it? When we run `KoalaTester`, it makes a call to the `main()` method of `Koala`, which prints the value of *count*. The purpose of all these examples is to show that `main()` can be called just like any other static method.

In addition to `main()` methods, static methods have two main purposes:

- For utility or helper methods that don't require any object state. Since there is no need to access instance variables, having static methods eliminates the need for the caller to instantiate the object just to call the method.
- For state that is shared by all instances of a class, like a counter. All instances must share the same state. Methods that merely use that state should be static as well.

Let's look at some examples covering other static concepts.

Calling a Static Variable or Method

Usually, accessing a static member is easy. You just put the classname before the method or variable and you are done. For example:

```
System.out.println(Koala.count);
Koala.main(new String[0]);
```

Both of these are nice and easy. There is one rule that is trickier. You can use an instance of the object to call a static method. The compiler checks for the type of the reference and uses that instead of the object—which is sneaky of Java. This code is perfectly legal:

```
5: Koala k = new Koala();
6: System.out.println(k.count);          // k is a Koala
7: k = null;
8: System.out.println(k.count);          // k is still a Koala
```

Believe it or not, this code outputs 0 twice. Line 6 sees that `k` is a `Koala` and `count` is a static variable, so it reads that static variable. Line 8 does the same thing. Java doesn't care that `k` happens to be `null`. Since we are looking for a static, it doesn't matter.

Remember to look at the reference type for a variable when you see a static method or variable. The exam creators will try to trick you into thinking a `NullPointerException` is thrown because the variable happens to be `null`. Don't be fooled!

One more time because this is really important: what does the following output?

```
Koala.count = 4;
Koala koala1 = new Koala();
Koala koala2 = new Koala();
koala1.count = 6;
koala2.count = 5;
System.out.println(Koala.count);
```

Hopefully, you answered 5. There is only one *count* variable since it is static. It is set to 4, then 6, and finally winds up as 5. All the `Koala` variables are just distractions.

Static vs. Instance

There's another way the exam creators will try to trick you regarding static and instance members. (Remember that "member" means field or method.) A static member cannot call an instance member. This shouldn't be a surprise since static doesn't require any instances of the class to be around.

The following is a common mistake for rookie programmers to make:

```
public class Static {
  private String name = "Static class";
  public static void first() {  }
  public static void second() {  }
  public void third() { System.out.println(name); }
  public static void main(String args[]) {
    first();
    second();
    third();          // DOES NOT COMPILE
  } }
```

The compiler will give you an error about making a static reference to a nonstatic method. If we fix this by adding `static` to `third()`, we create a new problem. Can you figure out what it is?

All this does is move the problem. Now, `third()` is referring to nonstatic `name`. Adding `static` to `name` as well would solve the problem. Another solution would have been to call `third` as an instance method—for example, `new Static().third();`.

The exam creators like this topic. A static method or instance method can call a static method because static methods don't require an object to use. Only an instance method can call another instance method on the same class without using a reference variable, because instance methods do require an object. Similar logic applies for the instance and static variables. Make sure you understand Table 4.3 before continuing.

TABLE 4.3 Static vs. instance calls

Type	Calling	Legal?	How?
Static method	Another static method or variable	Yes	Using the classname
Static method	An instance method or variable	No	
Instance method	A static method or variable	Yes	Using the classname or a reference variable
Instance method	Another instance method or variable	Yes	Using a reference variable

Let's try one more example so you have more practice at recognizing this scenario. Do you understand why the following lines fail to compile?

```
1:  public class Gorilla {
2:    public static int count;
3:    public static void addGorilla() { count++; }
4:    public void babyGorilla() { count++; }
5:    public void announceBabies() {
6:      addGorilla();
7:      babyGorilla();
8:    }
9:    public static void announceBabiesToEveryone() {
10:     addGorilla();
11:     babyGorilla();     // DOES NOT COMPILE
12:   }
13:   public int total;
14:   public static average = total / count;  // DOES NOT COMPILE
15: }
```

Lines 3 and 4 are fine because both static and instance methods can refer to a static variable. Lines 5–8 are fine because an instance method can call a static method. Line 11 doesn't compile because a static method cannot call an instance method. Similarly, line 14 doesn't compile because a static variable is trying to use an instance variable.

A common use for static variables is counting the number of instances:

```
public class Counter {
  private static int count;
  public Counter() { count++; }
```

```
  public static void main(String[] args) {
    Counter c1 = new Counter();
    Counter c2 = new Counter();
    Counter c3 = new Counter();
    System.out.println(count);          // 3
  }
}
```

Each time the constructor gets called, it increments *count* by 1. This example relies on the fact that static (and instance) variables are automatically initialized to the default value for that type, which is 0 for `int`. See Chapter 1 to review the default values.

Also notice that we didn't write `Counter.count`. We could have. It isn't necessary because we are already in that class so the compiler can infer it.

Static Variables

Some static variables are meant to change as the program runs. Counters are a common example of this. We want the count to increase over time. Just as with instance variables, you can initialize a static variable on the line it is declared:

```
public class Initializers {
  private static int counter = 0;          // initialization
}
```

Other static variables are meant to never change during the program. This type of variable is known as a *constant*. It uses the `final` modifier to ensure the variable never changes. `static final` constants use a different naming convention than other variables. They use all uppercase letters with underscores between "words." For example:

```
public class Initializers {
  private static final int NUM_BUCKETS = 45;
  public static void main(String[] args) {
    NUM_BUCKETS = 5;  // DOES NOT COMPILE
  } }
```

The compiler will make sure that you do not accidentally try to update a final variable. This can get interesting. Do you think the following compiles?

```
private static final ArrayList<String> values = new ArrayList<>();
public static void main(String[] args) {
  values.add("changed");
}
```

It actually does compile. *values* is a reference variable. We are allowed to call methods on reference variables. All the compiler can do is check that we don't try to reassign the `final` *values* to point to a different object.

Static Initialization

In Chapter 1, we covered instance initializers that looked like unnamed methods. Just code inside braces. Static initializers look similar. They add the `static` keyword to specify they should be run when the class is first used. For example:

```
private static final int NUM_SECONDS_PER_HOUR;
static {
  int numSecondsPerMinute = 60;
  int numMinutesPerHour = 60;
  NUM_SECONDS_PER_HOUR = numSecondsPerMinute * numMinutesPerHour;
}
```

The static initializer runs when the class is first used. The statements in it run and assign any static variables as needed. There is something interesting about this example. We just got through saying that final variables aren't allowed to be reassigned. The key here is that the static initializer is the first assignment. And since it occurs up front, it is okay.

Let's try another example to make sure you understand the distinction:

```
14: private static int one;
15: private static final int two;
16: private static final int three = 3;
17: private static final int four;     // DOES NOT COMPILE
18: static {
19:   one = 1;
20:   two = 2;
21:   three = 3;      // DOES NOT COMPILE
22:   two = 4;      // DOES NOT COMPILE
23: }
```

Line 14 declares a static variable that is not final. It can be assigned as many times as we like. Line 15 declares a final variable without initializing it. This means we can initialize it exactly once in a static block. Line 22 doesn't compile because this is the second attempt. Line 16 declares a final variable and initializes it at the same time. We are not allowed to assign it again, so line 21 doesn't compile. Line 17 declares a final variable that never gets initialized. The compiler gives a compiler error because it knows that the static blocks are the only place the variable could possibly get initialized. Since the programmer forgot, this is clearly an error.

Try to Avoid Static and Instance Initializers

Using static and instance initializers can make your code much harder to read. Everything that could be done in an instance initializer could be done in a constructor instead. The constructor approach is easier to read.

There is a common case to use a static initializer: when you need to initialize a static field and the code to do so requires more than one line. This often occurs when you want to initialize a collection like an `ArrayList`. When you do need to use a static initializer, put all the static initialization in the same block. That way, the order is obvious.

Static Imports

Back in Chapter 1, you saw that we could import a specific class or all the classes in a package:

```
import java.util.ArrayList;
import java.util.*;
```

We could use this technique to import:

```
import java.util.List;
import java.util.Arrays;
public class Imports {
  public static void main(String[] args) {
    List<String> list = Arrays.asList("one", "two");
  }
}
```

Imports are convenient because you don't need to specify where each class comes from each time you use it. There is another type of import called a static import. Regular imports are for importing classes. Static imports are for importing static members of classes. Just like regular imports, you can use a wildcard or import a specific member. The idea is that you shouldn't have to specify where each static method or variable comes from each time you use it. An example of when static interfaces shine are when you are referring to a lot of constants in another class.

In a large program, static imports can be overused. When importing from too many places, it can be hard to remember where each static member comes from.

The previous method has one static method call: `Arrays.asList`. Rewriting the code to use a static import yields the following:

```
import java.util.List;
import static java.util.Arrays.asList;          // static import
public class StaticImports {
  public static void main(String[] args) {
    List<String> list = asList("one", "two");     // no Arrays.
  } }
```

In this example, we are specifically importing the `asList` method. This means that any time we refer to `asList` in the class, it will call `Arrays.asList()`.

An interesting case is what would happen if we created an `asList` method in our `StaticImports` class. Java would give it preference over the imported one and the method we coded would be used.

The exam will try to trick you with misusing static imports. This example shows almost everything you can do wrong. Can you figure out what is wrong with each one?

```
1: import static java.util.Arrays; // DOES NOT COMPILE
2: import static java.util.Arrays.asList;
3: static import java.util.Arrays.*;  // DOES NOT COMPILE
4: public class BadStaticImports {
5:   public static void main(String[] args) {
6:    Arrays.asList("one");  // DOES NOT COMPILE
7:  } }
```

Line 1 tries to use a static import to import a class. Remember that static imports are only for importing static members. Regular imports are for importing a class. Line 3 tries to see if you are paying attention to the order of keywords. The syntax is `import static` and not vice versa. Line 6 is sneaky. We imported the `asList` method on line 2. However, we did not import the `Arrays` class anywhere. This makes it okay to write `asList("one");` but not `Arrays.asList("one");`.

There's only one more scenario with static imports. In Chapter 1, you learned that importing two classes with the same name gives a compiler error. This is true of static imports as well. The compiler will complain if you try to explicitly do a static import of two methods with the same name or two static variables with the same name. For example:

```
import static statics.A.TYPE;
import static statics.B.TYPE;     // DOES NOT COMPILE
```

Luckily when this happens, we can just refer to the static members via their classname in the code instead of trying to use a static import.

Passing Data Among Methods

Java is a "pass-by-value" language. This means that a copy of the variable is made and the method receives that copy. Assignments made in the method do not affect the caller. Let's look at an example:

```
2: public static void main(String[] args) {
3:   int num = 4;
4:   newNumber(5);
```

```
5:   System.out.println(num);     // 4
6: }
7: public static void newNumber(int num) {
8:   num = 8;
9: }
```

On line 3, `num` is assigned the value of 4. On line 4, we call a method. On line 8, the `num` parameter in the method gets set to 8. Although this parameter has the same name as the variable on line 3, this is a coincidence. The name could be anything. The exam will often use the same name to try to confuse you. The variable on line 3 never changes because no assignments are made to it.

Now that you've seen primitives, let's try an example with a reference type. What do you think is output by the following code?

```
public static void main(String[] args) {
  String name = "Webby";
  speak(name);
  System.out.println(name);
}
public static void speak(String name) {
  name = "Sparky";
}
```

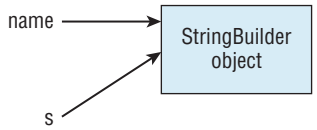
The correct answer is `Webby`. Just as in the primitive example, the variable assignment is only to the method parameter and doesn't affect the caller.

Notice how we keep talking about variable assignments. This is because we can call methods on the parameters. As an example, we have code that calls a method on the `StringBuilder` passed into the method:

```
public static void main(String[] args) {
  StringBuilder name = new StringBuilder();
  speak(name);
  System.out.println(name); // Webby
}
public static void speak(StringBuilder s) {
  s.append("Webby");
}
```

In this case, the output is `Webby` because the method merely calls a method on the parameter. It doesn't reassign *name* to a different object. In Figure 4.4, you can see how pass-by-value is still used. `s` is a copy of the variable *name*. Both point to the same `StringBuilder`, which means that changes made to the `StringBuilder` are available to both references.

FIGURE 4.4 Copying a reference with pass-by-value



Real World Scenario

Pass-by-Value vs. Pass-by-Reference

Different languages handle parameters in different ways. Pass-by-value is used by many languages, including Java. In this example, the swap method does not change the original values. It only changes *a* and *b* within the method.

```
public static void main(String[] args) {
    int original1 = 1;
    int original2 = 2;
    swap(original1, original2);
    System.out.println(original1);      // 1
    System.out.println(original2);      // 2
}
public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

The other approach is pass-by-reference. It is used by default in a few languages, such as Perl. We aren't going to show you Perl code here because you are studying for the Java exam and we don't want to confuse you. The following example is in a made-up language that shows pass-by-reference:

```
original1 = 1;
original2 = 2;
swapByReference(original1, original2);
print(original1);       // 2 (not in Java)
print(original2);       // 1 (not in Java)

swapByReference(a, b) {
  temp = a;
  a = b;
  b = temp;
}
```

See the difference? In our made-up language, the caller is affected by variable assignments made in the method.

To review, Java uses pass-by-value to get data into a method. Assigning a new primitive or reference to a parameter doesn't change the caller. Calling methods on a reference to an object does affect the caller.

Getting data back from a method is easier. A copy is made of the primitive or reference and returned from the method. Most of the time, this returned value is used. For example, it might be stored in a variable. If the returned value is not used, the result is ignored. Watch for this on the exam. Ignored returned values are tricky.

Let's try an example. Pay attention to the return types.

```
1: public class ReturningValues {
2:  public static void main(String[] args) {
3:    int number = 1;                          // 1
4:    String letters = "abc";                  // abc
5:    number(number);                          // 1
6:    letters = letters(letters);              // abcd
7:    System.out.println(number + letters);    // 1abcd
8:  }
9:  public static int number(int number) {
10:    number++;
11:    return number;
12:  }
13:  public static String letters(String letters) {
14:    letters += "d";
15:    return letters;
16:  }
17: }
```

This is a tricky one because there is a lot to keep track of. When you see such questions on the exam, write down the values of each variable. Lines 3 and 4 are straightforward assignments. Line 5 calls a method. Line 10 increments the method parameter to 2 but leaves the *numbers* variable in the `main()` method as 1. While line 11 returns the value, the caller ignores it. The method call on line 6 doesn't ignore the result so *letters* becomes `"abcd"`. Remember that this is happening because of the returned value and not the method parameter.

Overloading Methods

Now that you are familiar with the rules for declaring methods, it is time to look at creating methods with the same signature in the same class. *Method overloading* occurs when there are different method signatures with the same name but different type parameters.

We've been calling overloaded methods for a while. `System.out.println` and `StringBuilder`'s `append` methods provide many overloaded versions so you can pass just about anything to them without having to think about it. In both of these examples, the

only change was the type of the parameter. Overloading also allows different numbers of parameters.

Everything other than the method signature can vary for overloaded methods. This means there can be different access modifiers, specifiers (like static), return types, and exception lists.

These are all valid overloaded methods:

```
public void fly(int numMiles) { }
public void fly(short numFeet) { }
public boolean fly() { return false; }
void fly(int numMiles, short numFeet) { }
public void fly(short numFeet, int numMiles) throws Exception { }
```

As you can see, we can overload by changing anything in the parameter list. We can have a different type, more types, or the same types in a different order. Also notice that the access modifier and exception list are irrelevant to overloading.

Now let's look at an example that is not valid overloading:

```
public void fly(int numMiles) { }
public int fly(int numMiles) { }     // DOES NOT COMPILE
```

This method doesn't compile because it only differs from the original by return type. The parameter lists are the same so they are duplicate methods as far as Java is concerned.

What about these two? Why does the second not compile?

```
public void fly(int numMiles) { }
public static void fly(int numMiles) { }     // DOES NOT COMPILE
```

Again, the parameter list is the same. The only difference is that one is an instance method and one is a static method.

Calling overloaded methods is easy. You just write code and Java calls the right one. For example, look at these two methods:

```
public void fly(int numMiles) {
  System.out.println("short");
}
public void fly(short numFeet) {
  System.out.println("short");
}
```

The call `fly((short) 1);` prints short. It looks for matching types and calls the appropriate method. Of course, it can be more complicated than this.

Now that you know the basics of overloading, let's look at some more complex scenarios that you may encounter on the exam.

Overloading and Varargs

Which method do you think is called if we pass an `int[]`?

```
public void fly(int[] lengths) { }
public void fly(int... lengths) { }     // DOES NOT COMPILE
```

Trick question! Remember that Java treats varargs as if they were an array. This means that the method signature is the same for both methods. Since we are not allowed to overload methods with the same parameter list, this code doesn't compile. Even though the code doesn't look the same, it compiles to the same parameter list.

Now that we've just gotten through explaining that they are the same, it is time to mention how they are not the same. It shouldn't be a surprise that you can call either method by passing an array:

```
fly(new int[] { 1, 2, 3 });
```

However, you can only call the varargs version with stand-alone parameters:

```
fly(1, 2, 3);
```

Obviously, this means they don't compile *exactly* the same. The parameter list is the same, though, and that is what you need to know with respect to overloading for the exam.

Autoboxing

In the previous chapter, you saw how Java will convert a primitive `int` to an object `Integer` to add it to an `ArrayList` through the wonders of autoboxing. This works for code you write too.

```
public void fly(Integer numMiles) { }
```

This means calling `fly(3);` will call the previous method as expected. However, what happens if we have both a primitive and an integer version?

```
public void fly(int numMiles) { }
public void fly(Integer numMiles) { }
```

Java will match the `int numMiles` version. Java tries to use the most specific parameter list it can find. When the primitive `int` version isn't present, it will autobox. However, when the primitive `int` version is provided, there is no reason for Java to do the extra work of autoboxing.

Reference Types

Given the rule about Java picking the most specific version of a method that it can, what do you think this code outputs?

```
public class ReferenceTypes {
  public void fly(String s) {
    System.out.print("string ");
  }

  public void fly(Object o) {
    System.out.print("object ");
  }
  public static void main(String[] args) {
    ReferenceTypes r = new ReferenceTypes();
```

```
    r.fly("test");
    r.fly(56);
  } }
```

The answer is "string object". The first call is a String and finds a direct match. There's no reason to use the Object version when there is a nice String parameter list just waiting to be called. The second call looks for an int parameter list. When it doesn't find one, it autoboxes to Integer. Since it still doesn't find a match, it goes to the Object one.

Primitives

Primitives work in a way similar to reference variables. Java tries to find the most specific matching overloaded method. What do you think happens here?

```
public class Plane {
  public void fly(int i) {
    System.out.print("int ");
  }
  public void fly(long l) {
    System.out.print("long ");
  }
  public static void main(String[] args) {
     Plane p = new Plane();
     p.fly(123);
     p.fly(123L);
  } }
```

The answer is int long. The first call passes an int and sees an exact match. The second call passes a long and also sees an exact match. If we comment out the overloaded method with the int parameter list, the output becomes long long. Java has no problem calling a larger primitive. However, it will not do so unless a better match is not found.

Note that Java can only accept wider types. An int can be passed to a method taking a long parameter. Java will not automatically convert to a narrower type. If you want to pass a long to a method taking an int parameter, you have to add a cast to explicitly say narrowing is okay.

Putting It All Together

So far, all the rules for when an overloaded method is called should be logical. Java calls the most specific method it can. When some of the types interact, the Java rules focus on backward compatibility. In Java 1.4 and earlier, autoboxing and varargs didn't exist. Although that was a long time ago, old code still needs to work—which means autoboxing and varargs come last when Java looks at overloaded methods. Ready for the official order? Table 4.4 lays it out for you.

TABLE 4.4 Order Java uses to choose the right overloaded method

Rule	Example of what will be chosen for `glide(1,2)`
Exact match by type	`public String glide(int i, int j) {}`
Larger primitive type	`public String glide(long i, long j) {}`
Autoboxed type	`public String glide(Integer i, Integer j) {}`
Varargs	`public String glide(int... nums) {}`

Let's give this a practice run using the rules in Table 4.4. What do you think this outputs?

```
public class Glider2 {
  public static String glide(String s) {
    return "1";
  }
  public static String glide(String... s) {
    return "2";
  }
  public static String glide(Object o) {
    return "3";
  }
  public static String glide(String s, String t) {
    return "4";
  }
  public static void main(String[] args) {
    System.out.print(glide("a"));
    System.out.print(glide("a", "b"));
    System.out.print(glide("a", "b", "c"));
  } }
```

It prints out `142`. The first call matches the signature taking a single `String` because that is the most specific match. The second call matches the signature, taking two `String` parameters since that is an exact match. It isn't until the third call that the varargs version is used since there are no better matches.

As accommodating as Java is with trying to find a match, it will only do one conversion:

```
public class TooManyConversions {
  public static void play(Long l) { }
```

```
public static void play(Long... l) { }
public static void main(String[] args) {
  play(4);     // DOES NOT COMPILE
  play(4L);     // calls the Long version
} }
```

Here we have a problem. Java is happy to convert the `int` 4 to a `long` 4 or an `Integer` 4. It cannot handle converting in two steps to a `long` and then to a `Long`. If we had `public static void play(Object o) { }`, it would match because only one conversion would be necessary: from `int` to `Integer`. An `Integer` is an `Object`, as you'll see in Chapter 5.

Creating Constructors

As you learned in Chapter 1, a constructor is a special method that matches the name of the class and has no return type. Here's an example:

```
public class Bunny {
  public Bunny() {
    System.out.println("constructor");
  }
}
```

The name of the constructor, `Bunny`, matches the name of the class, `Bunny`, and there is no return type, not even `void`. That makes this a constructor. Can you tell why these two are not valid constructors for the `Bunny` class?

```
public bunny() { }     // DOES NOT COMPILE
public void Bunny() { }
```

The first one doesn't match the classname because Java is case sensitive. Since it doesn't match, Java knows it can't be a constructor and is supposed to be a regular method. However, it is missing the return type and doesn't compile. The second method is a perfectly good method, but is not a constructor because it has a return type.

Constructors are used when creating a new object. This process is called *instantiation* because it creates a new instance of the class. A constructor is called when we write `new` followed by the name of the class we want to instantiate. For example:

```
new Bunny()
```

When Java sees the `new` keyword, it allocates memory for the new object. Java also looks for a constructor and calls it.

A constructor is typically used to initialize instance variables. The `this` keyword tells Java you want to reference an instance variable. Most of the time, `this` is optional. The problem is that sometimes there are two variables with the same name. In a constructor, one is a parameter and one is an instance variable. If you don't say otherwise, Java gives

you the one with the most granular scope, which is the parameter. Using `this.name` tells Java you want the instance variable.

Here's a common way of writing a constructor:

```
1: public class Bunny {
2:   private String color;
3:   public Bunny(String color) {
4:     this.color = color;
5:  } }
```

On line 4, we assign the parameter `color` to the instance variable *color*. The right side of the assignment refers to the parameter because we don't specify anything special. The left side of the assignment uses `this` to tell Java we want it to use the instance variable.

Now let's look at some examples that aren't common but that you might see on the exam:

```
1: public class Bunny {
2:   private String color;
3:   private int height;
4:   private int length;
5:   public Bunny(int length, int theHeight) {
6:     length = this.length;     // backwards - no good!
7:     height = theHeight;          // fine because a different name
8:     this.color = "white";     // fine, but redundant
9:   }
10: public static void main(String[] args) {
11:  Bunny b = new Bunny(1, 2);
12:  System.out.println(b.length + " " + b.height + " " + b.color);
13:  } }
```

Line 6 is incorrect and you should watch for it on the exam. The instance variable *length* starts out with a 0 value. That 0 is assigned to the method parameter *length*. The instance variable stays at 0. Line 7 is more straightforward. The parameter *theHeight* and instance variable *height* have different names. Since there is no naming collision, `this` is not required. Finally, line 8 shows that it is allowed to use `this` even when there is no duplication of variable names.

In this section, we'll look at default constructors, overloading constructors, final fields, and the order of initialization in a class.

Default Constructor

Every class in Java has a constructor whether you code one or not. If you don't include any constructors in the class, Java will create one for you without any parameters.

This Java-created constructor is called the *default constructor*. Sometimes we call it the default no-arguments constructor for clarity. Here's an example:

```
public class Rabbit {
  public static void main(String[] args) {
    Rabbit rabbit = new Rabbit();          // Calls default constructor
  }
}
```

In the `Rabbit` class, Java sees no constructor was coded and creates one. This default constructor is equivalent to typing this:

```
public Rabbit() {}
```

The default constructor has an empty parameter list and an empty body. It is fine for you to type this in yourself. However, since it doesn't do anything, Java is happy to supply it for you and save you some typing.

We keep saying *generated*. This happens during the compile step. If you look at the file with the `.java` extension, the constructor will still be missing. It is only in the compiled file with the `.class` extension that it makes an appearance.

Remember that a default constructor is only supplied if there are no constructors present. Which of these classes do you think has a default constructor?

```
class Rabbit1 {
}
class Rabbit2 {
  public Rabbit2() { }
}
class Rabbit3 {
  public Rabbit3(boolean b) { }
}
class Rabbit4 {
  private Rabbit4() { }
}
```

Only `Rabbit1` gets a default no-argument constructor. It doesn't have a constructor coded so Java generates a default no-argument constructor. `Rabbit2` and `Rabbit3` both have public constructors already. `Rabbit4` has a private constructor. Since these three classes have a constructor defined, the default no-argument constructor is not inserted for you.

Let's take a quick look at how to call these constructors:

```
1: public class RabbitsMultiply {
2:   public static void main(String[] args) {
```

```
3:    Rabbit1 r1 = new Rabbit1();
4:    Rabbit2 r2 = new Rabbit2();
5:    Rabbit3 r3 = new Rabbit3(true);
6:    Rabbit4 r4 = new Rabbit4(); // DOES NOT COMPILE
7:   } }
```

Line 3 calls the generated default no-argument constructor. Lines 4 and 5 call the user-provided constructors. Line 6 does not compile. `Rabbit4` made the constructor private so that other classes could not call it.

Having a private constructor in a class tells the compiler not to provide a default no-argument constructor. It also prevents other classes from instantiating the class. This is useful when a class only has static methods or the class wants to control all calls to create new instances of itself.

Overloading Constructors

Up to now, you've only seen one constructor per class. You can have multiple constructors in the same class as long as they have different method signatures. When overloading methods, the method name and parameter list needed to match. With constructors, the name is always the same since it has to be the same as the name of the class. This means constructors must have different parameters in order to be overloaded.

This example shows two constructors:

```
public class Hamster {
  private String color;
  private int weight;
  public Hamster(int weight) {               // first constructor
    this.weight = weight;
    color = "brown";
  }
  public Hamster(int weight, String color) {     // second constructor
   this.weight = weight;
   this.color = color;
  }
}
```

One of the constructors takes a single `int` parameter. The other takes an `int` and a `String`. These parameter lists are different, so the constructors are successfully overloaded.

There is a problem here, though. There is a bit of duplication. In programming, even a bit of duplication tends to turn into a lot of duplication as we keep adding "just one more

thing." What we really want is for the first constructor to call the second constructor with two parameters. You might be tempted to write this:

```
public Hamster(int weight) {
  Hamster(weight, "brown");     // DOES NOT COMPILE
}
```

This will not work. Constructors can be called only by writing `new` before the name of the constructor. They are not like normal methods that you can just call. What happens if we stick `new` before the constructor name?

```
public Hamster(int weight) {
  new Hamster(weight, "brown");     // Compiles but does not do what we want
}
```

This attempt does compile. It doesn't do what we want, though. When the constructor with one parameter is called, it creates an object with the default `weight` and `color`. It then constructs a different object with the desired `weight` and `color` and ignores the new object. That's not what we want. We want `weight` and `color` set on the object we are trying to instantiate in the first place.

Java provides a solution: `this`—yes, the same keyword we used to refer to instance variables. When `this` is used as if it were a method, Java calls another constructor on the same instance of the class.

```
public Hamster(int weight) {
  this(weight, "brown");
}
```

Success! Now Java calls the constructor that takes two parameters. `weight` and `color` get set on this instance.

`this()` has one special rule you need to know. If you choose to call it, the `this()` call must be the first noncommented statement in the constructor.

```
3: public Hamster(int weight) {
4:   System.out.println("in constructor");
5:   // ready to call this
6:   this(weight, "brown");     // DOES NOT COMPILE
7: }
```

Even though a print statement on line 4 doesn't change any variables, it is still a Java statement and is not allowed to be inserted before the call to `this()`. The comment on line 5 is just fine. Comments don't run Java statements and are allowed anywhere.

Real World Scenario

Constructor Chaining

Overloaded constructors often call each other. One common technique is to have each constructor add one parameter until getting to the constructor that does all the work. This approach is called *constructor chaining*. In this example, all three constructors are chained.

```
public class Mouse {
   private int numTeeth;
   private int numWhiskers;
   private int weight;

   public Mouse(int weight) {
     this(weight, 16);  // calls constructor with 2 parameters
   }
   public Mouse(int weight, int numTeeth) {
    this(weight, numTeeth, 6);  // calls constructor with 3 parameters
   }
   public Mouse(int weight, int numTeeth, int numWhiskers) {
     this.weight = weight;
     this.numTeeth = numTeeth;
     this.numWhiskers = numWhiskers;
   }
   public void print() {
     System.out.println(weight + " " + numTeeth + " " + numWhiskers);
   }
   public static void main(String[] args) {
     Mouse mouse = new Mouse(15);
     mouse.print();
   }
}
```

This code prints `15 16 6`. The `main()` method calls the constructor with one parameter. That constructor adds a second hard-coded value and calls the constructor with two parameters. That constructor adds one more hard-coded value and calls the constructor with three parameters. The three-parameter constructor assigns the instance variables.

Final Fields

As you saw earlier in the chapter, `final` instance variables must be assigned a value exactly once. We saw this happen in the line of the declaration and in an instance initializer. There is one more location this assignment can be done: in the constructor.

```
public class MouseHouse {
  private final int volume;
  private final String name = "The Mouse House";
  public MouseHouse(int length, int width, int height) {
   volume = length * width * height;
  }}
```

The constructor is part of the initialization process, so it is allowed to assign `final` instance variables in it. By the time the constructor completes, all `final` instance variables must have been set.

Order of Initialization

Chapter 1 covered the order of initialization. Now that you've learned about static initializers, it is time to revisit that. Unfortunately, you do have to memorize this list. We'll give you lots of practice, but you do need to know this order by heart.

1. If there is a superclass, initialize it first (we'll cover this rule in the next chapter. For now, just say "no superclass" and go on to the next rule.)
2. Static variable declarations and static initializers in the order they appear in the file.
3. Instance variable declarations and instance initializers in the order they appear in the file.
4. The constructor.

Let's try the first example:

```
1: public class InitializationOrderSimple {
2:   private String name = "Torchie";
3:   { System.out.println(name); }
4:   private static int COUNT = 0;
5:   static { System.out.println(COUNT); }
6:   static { COUNT += 10; System.out.println(COUNT); }
7:   public InitializationOrderSimple() {
8:     System.out.println("constructor");
9:   } }

1: public class CallInitializationOrderSimple {
2:   public static void main(String[] args) {
```

```
3:     InitializationOrderSimple init = new InitializationOrderSimple();
4:   } }
```

The output is:

```
0
10
Torchie
constructor
```

Let's look at why. Rule 1 doesn't apply because there is no superclass. Rule 2 says to run the `static` variable declarations and `static` initializers—in this case, lines 5 and 6, which output `0` and `10`. Rule 3 says to run the instance variable declarations and instance initializers—here, lines 2 and 3, which output `Torchie`. Finally, rule 4 says to run the constructor—here, lines 7–9, which output `constructor`.

The next example is a little harder. Keep in mind that the four rules apply only if an object is instantiated. If the class is referred to without a `new` call, only rules 1 and 2 apply. The other two rules relate to instances and constructors. They have to wait until there is code to instantiate the object.

What do you think happens here?

```
1: public class InitializationOrder {
2:   private String name = "Torchie";
3:   { System.out.println(name); }
4:   private static int COUNT = 0;
5:   static { System.out.println(COUNT); }
6:   { COUNT++;  System.out.println(COUNT); }
7:   public InitializationOrder() {
8:     System.out.println("constructor");
9:   }
10:  public static void main(String[] args) {
11:    System.out.println("read to construct");
12:    new InitializationOrder();
13:  }
14: }
```

The output looks like this:

```
0
read to construct
Torchie
1
constructor
```

Again, rule 1 doesn't apply because there is no superclass. Rule 2 tells us to look at the `static` variables and `static` initializers—lines 4 and 5, in that order. Line 5 outputs `0`. Now that the statics are out of the way, the `main()` method can run. Next, we can use rule 3 to run the instance variables and instance initializers. Here that is lines 2 and 3, which output `Torchie`. Finally, rule 4 says to run the constructor—in this case, lines 7–9, which output `constructor`.

We are going to try one more example. This one is as hard as it gets. If you understand the output of this next one, congratulations on a job well done; if not, don't worry. Write some programs to play with this. Try typing in the examples in this section and making minor changes to see what happens. For example, you might try commenting out part of the code. This will give you a better feel for what is going on. Then come back and reread this section to try the examples.

Ready for the tough example? Here it is:

```
1: public class YetMoreInitializationOrder {
2:   static { add(2); }
3:   static void add(int num) { System.out.print(num + " "); }
4:   YetMoreInitializationOrder() { add(5); }
5:   static { add(4); }
6:   { add(6); }
7:   static { new YetMoreInitializationOrder(); }
8:   { add(8); }
9:   public static void main(String[] args) { } }
```

The correct answer is `2 4 6 8 5`. Let's walk through why that is. There is no superclass, so we jump right to rule 2—the statics. There are three `static` blocks: on lines 2, 5, and 7. They run in that order. The `static` block on line 2 calls the `add()` method, which prints `2`. The `static` block on line 5 calls the `add()` method, which prints `4`. The last `static` block, on line 7, calls `new` to instantiate the object. This means we can go on to rule 3 to look at the instance variables and instance initializers. There are two of those: on lines 6 and 8. They both call the `add()` method and print `6` and `8`, respectively. Finally, we go on to rule 4 and call the constructor, which calls the `add()` method one more time and prints `5`.

This example is tricky for a few reasons. There's a lot to keep track of. Also, the question has a lot of one-line code blocks and methods, making it harder to visualize which is a block. Luckily, questions like this are rare on the exam. If you see one, just write down what is going on as you read the code.

Encapsulating Data

In Chapter 1, we had an example of a class with a field that wasn't private:

```
public class Swan {
  int numberEggs;     // instance variable
}
```

Why do we care? Since there is default (package private) access, that means any class in the package can set *numberEggs*. We no longer have control of what gets set in our own class. A caller could even write this:

```
mother.numberEggs = -1;
```

This is clearly no good. We do not want the mother `Swan` to have a negative number of eggs!

Encapsulation to the rescue. Encapsulation means we set up the class so only methods in the class with the variables can refer to the instance variables. Callers are required to use these methods. Let's take a look at our newly encapsulated `Swan` class:

```
1: public class Swan {
2:  private int numberEggs;                    // private
3:  public int getNumberEggs() {                    // getter
4:    return numberEggs;
5:  }
6:  public void setNumberEggs(int numberEggs) {     // setter
7:    if (numberEggs >= 0)                     // guard condition
8:      this.numberEggs = numberEggs;
9:  } }
```

Note *numberEggs* is now private on line 2. This means only code within the class can read or write the value of *numberEggs*. Since we wrote the class, we know better than to set a negative number of eggs. We added a method on lines 3–5 to read the value, which is called an *accessor method* or a getter. We also added a method on lines 6–9 to update the value, which is called a *mutator method* or a setter. The setter has an `if` statement in this example to prevent setting the instance variable to an invalid value. This guard condition protects the instance variable.

On line 8, we used the `this` keyword that we saw in constructors to differentiate between the method parameter *numberEggs* and the instance variable *numberEggs*.

For encapsulation, remember that data (an instance variable) is private and getters/setters are public. Java defines a naming convention that is used in *JavaBeans*. JavaBeans are reusable software components. JavaBeans call an instance variable a *property*. The only thing you need to know about JavaBeans for the exam is the naming conventions listed in Table 4.5.

TABLE 4.5 Rules for JavaBeans naming conventions

Rule	Example
Properties are private.	`private int numEggs;`

Getter methods begin with `is` if the property is a `boolean`.	`public boolean isHappy() {`
	`  return happy;`
	`}`

Getter methods begin with `get` if the property is not a `boolean`.	`public int getNumEggs() {`
	`  return numEggs;`
	`}`

Setter methods begin with `set`.	`public void setHappy(boolean happy) {`
	`  this.happy = happy;`
	`}`

The method name must have a prefix of `set`/`get`/`is`, followed by the first letter of the property in uppercase, followed by the rest of the property name.	`public void setNumEggs(int num) {`
	`  numEggs = num;`
	`}`

From the last example in Table 4.5, you noticed that you can name the method parameter to set anything you want. Only the method name and property name have naming conventions here.

It's time for some practice. See if you can figure out which lines follow JavaBeans naming conventions:

```
12: private boolean playing;
13: private String name;
14: public boolean getPlaying() { return playing; }
15: public boolean isPlaying() { return playing; }
16: public String name() { return name; }
17: public void updateName(String n) { name = n; }
18: public void setname(String n) { name = n; }
```

Lines 12 and 13 are good. They are private instance variables. Line 14 doesn't follow the JavaBeans naming conventions. Since *`playing`* is a `boolean`, the getter must begin with `is`. Line 15 is a correct getter for *`playing`*. Line 16 doesn't follow the JavaBeans naming conventions because it should be called `getName`. Lines 17 and 18 do not follow the JavaBeans naming conventions because they should be named `setName`. Remember that Java is case sensitive, so `setname` is not adequate to meet the naming convention.

Creating Immutable Classes

Encapsulating data is helpful because it prevents callers from making uncontrolled changes to your class. Another common technique is making classes immutable so they cannot be changed at all.

Immutable classes are helpful because you know they will always be the same. You can pass them around your application with a guarantee that the caller didn't change anything. This helps make programs easier to maintain. It also helps with performance by limiting the number of copies, as you saw with `String` in Chapter 3, "Core Java APIs."

One step in making a class immutable is to omit the setters. But wait: we still want the caller to be able to specify the initial value—we just don't want it to change after the object is created. Constructors to the rescue!

```
public class ImmutableSwan {
  private int numberEggs;
  public ImmutableSwan(int numberEggs) {
    this.numberEggs = numberEggs;
  }
  public int getNumberEggs() {
    return numberEggs;
  } }
```

In this example, we don't have a setter. We do have a constructor that allows a value to be set. Remember, immutable is only measured after the object is constructed. Immutable classes are allowed to have values. They just can't change after instantiation.

Real World Scenario

Return Types in Immutable Classes

When you are writing an immutable class, be careful about the return types. On the surface, this class appears to be immutable since there is no setter:

```
public class NotImmutable {
   private StringBuilder builder;
   public NotImmutable(StringBuilder b) {
     builder = b;
   }
   public StringBuilder getBuilder() {
     return builder;
   } }
```

The problem is that it isn't really. Consider this code snippet:

continues

continued

```
StringBuilder sb = new StringBuilder("initial");
NotImmutable problem = new NotImmutable(sb);
sb.append(" added");
StringBuilder gotBuilder = problem.getBuilder();
gotBuilder.append(" more");
System.out.println(problem.getBuilder());
```

This outputs `"initial added more"`—clearly not what we were intending. The problem is that we are just passing the same `StringBuilder` all over. The caller has a reference since it was passed to the constructor. Anyone who calls the getter gets a reference too.
A solution is to make a copy of the mutable object. This is called a defensive copy.

```
public Mutable(StringBuilder b) {
  builder = new StringBuilder(b);
}
public StringBuilder getBuilder() {
  return new StringBuilder(builder);
}
```

Now the caller can make changes to the initial `sb` object and it is fine. `Mutable` no longer cares about that object after the constructor gets run. The same goes for the getter: callers can change their `StringBuilder` without affecting `Mutable`.

Another approach for the getter is to return an immutable object:

```
public String getValue() {
  return builder.toString();
}
```

There's no rule that says we have to return the same type as we are storing. `String` is safe to return because it is immutable in the first place.

To review, encapsulation refers to preventing callers from changing the instance variables directly. Immutability refers to preventing callers from changing the instance variables at all.

Writing Simple Lambdas

Java is an object-oriented language at heart. You've seen plenty of objects by now. In Java 8, the language added the ability to write code using another style. *Functional programming* is a way of writing code more declaratively. You specify what you want to do rather than dealing with the state of objects. You focus more on expressions than loops.

Functional programming uses lambda expressions to write code. A *lambda expression* is a block of code that gets passed around. You can think of a lambda expression as an

anonymous method. It has parameters and a body just like full-fledged methods do, but it doesn't have a name like a real method. Lambda expressions are often referred to as lambdas for short. You might also know them as closures if Java isn't your first language. If you had a bad experience with closures in the past, don't worry. They are far simpler in Java.

In other words, a lambda expression is like a method that you can pass as if it were a variable. For example, there are different ways to calculate age. One human year is equivalent to seven dog years. You want to write a method that that takes an `age()` method as input. To do this in an object-oriented program, you'd need to define a `Human` subclass and a `Dog` subclass. With lambdas, you can just pass in the relevant expression to calculate age.

Lambdas allow you to write powerful code in Java. Only the simplest lambda expressions are on the OCA exam. The goal is to get you comfortable with the syntax and the concepts. This means you aren't truly doing functional programming yet. You'll see lambdas again on the OCP exam.

In this section, we'll cover an example of why lambdas are helpful, the syntax of lambdas, and how to use predicates.

Lambda Example

Our goal is to print out all the animals in a list according to some criteria. We'll show you how to do this without lambdas to illustrate how lambdas are useful. We start out with the `Animal` class:

```
public class Animal {
  private String species;
  private boolean canHop;
  private boolean canSwim;
  public Animal(String speciesName, boolean hopper, boolean swimmer) {
    species = speciesName;
    canHop = hopper;
    canSwim = swimmer;
  }
   public boolean canHop() { return canHop; }
   public boolean canSwim() { return canSwim; }
   public String toString() { return species; }
}
```

The `Animal` class has three instance variables, which are set in the constructor. It has two methods that get the state of whether the animal can hop or swim. It also has a `toString()` method so we can easily identify the `Animal` in programs.

We plan to write a lot of different checks, so we want an interface. You'll learn more about interfaces in the next chapter. For now, it is enough to remember that an interface specifies the methods that our class needs to implement:

```
public interface CheckTrait {
  boolean test(Animal a);
}
```

The first thing we want to check is whether the `Animal` can hop. We provide a class that can check this:

```
public class CheckIfHopper implements CheckTrait {
  public boolean test(Animal a) {
    return a.canHop();
  }
}
```

This class may seem simple—and it is. This is actually part of the problem that lambdas solve. Just bear with us for a bit. Now we have everything that we need to write our code to find the `Animals` that hop:

```
1: public class TraditionalSearch {
2:   public static void main(String[] args) {
3:     List<Animal> animals = new ArrayList<Animal>();  // list of animals
4:     animals.add(new Animal("fish", false, true));
5:     animals.add(new Animal("kangaroo", true, false));
6:     animals.add(new Animal("rabbit", true, false));
7:     animals.add(new Animal("turtle", false, true));
8:
9:     print(animals, new CheckIfHopper());      // pass class that does check
10:   }
11:  private static void print(List<Animal> animals, CheckTrait checker) {
12:     for (Animal animal : animals) {
13:       if (checker.test(animal))               // the general check
14:         System.out.print(animal + " ");
15:     }
16:     System.out.println();
17:   }
18: }
```

The `print()` method on line 11 method is very general—it can check for any trait. This is good design. It shouldn't need to know what specifically we are searching for in order to print a list of animals.

Now what happens if we want to print the `Animals` that swim? Sigh. We need to write another class `CheckIfSwims`. Granted, it is only a few lines. Then we need to add a new line under line 9 that instantiates that class. That's two things just to do another check.

Why can't we just specify the logic we care about right here? Turns out that we can with lambda expressions. We could repeat that whole class here and make you find the one line that changed. Instead, we'll just show you. We could replace line 9 with the following, which uses a lambda:

```
9:    print(animals, a -> a.canHop());
```

Don't worry that the syntax looks a little funky. You'll get used to it and we'll describe it in the next section. We'll also explain the bits that look like magic. For now, just focus on how easy it is to read. We are telling Java that we only care about `Animals` that can hop.

It doesn't take much imagination to figure how we would add logic to get the `Animals` that can swim. We only have to add one line of code—no need for an extra class to do something simple. Here's that other line:

```
print(animals, a -> a.canSwim());
```

How about `Animals` that cannot swim?

```
print(animals, a -> ! a.canSwim());
```

The point here is that it is really easy to write code that uses lambdas once you get the basics in place. This code uses a concept called deferred execution. *Deferred execution* means that code is specified now but will run later. In this case, later is when the `print()` method calls it.

Lambda Syntax

One of the simplest lambda expressions you can write is the one you just saw:

```
a -> a.canHop();
```

This means that Java should call a method with an `Animal` parameter that returns a `boolean` value that's the result of `a.canHop()`. We know all this because we wrote the code. But how does Java know?

Java replies on context when figuring out what lambda expressions mean. We are passing this lambda as the second parameter of the `print()` method. That method expects a `CheckTrait` as the second parameter. Since we are passing a lambda instead, Java tries to map our lambda to that interface:

```
boolean test(Animal a);
```

Since that interface's method takes an `Animal`, that means the lambda parameter has to be an `Animal`. And since that interface's method returns a `boolean`, we know the lambda returns a `boolean`.

The syntax of lambdas is tricky because many parts are optional. These two lines do the exact same thing:

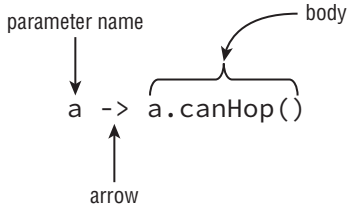
```
a -> a.canHop()

(Animal a) -> { return a.canHop(); }
```

Let's look at what is going on here. The first example, shown in Figure 4.5, has three parts:

- Specify a single parameter with the name *a*
- The arrow operator to separate the parameter and body
- A body that calls a single method and returns the result of that method

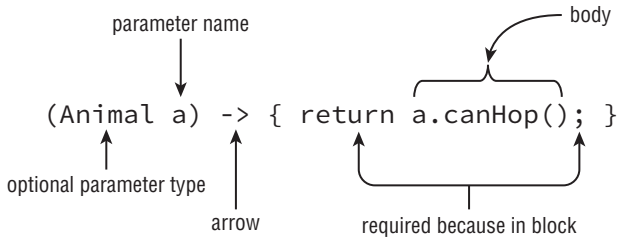
FIGURE 4.5 Lambda syntax omitting optional parts



The second example also has three parts; it's just more verbose (see Figure 4.6):

- Specify a single parameter with the name *a* and stating the type is `Animal`
- The arrow operator to separate the parameter and body
- A body that has one or more lines of code, including a semicolon and a return statement

FIGURE 4.6 Lambda syntax, including optional parts



The parentheses can only be omitted if there is a single parameter and its type is not explicitly stated. Java does this because developers commonly use lambda expressions this way and they can do as little typing as possible.

It shouldn't be news to you that we can omit braces when we only have a single statement. We did this with `if` statements and loops already. What is different here is that the rules change when you omit the braces. Java doesn't require you to type `return` or use a semicolon when no braces are used. This special shortcut doesn't work when we have two or more statements. At least this is consistent with using `{}` to create blocks of code elsewhere.

Let's look at some examples of valid lambdas. Pretend that there are valid interfaces that can consume a lambda with zero, one, or two `String` parameters.

```
3: print(() -> true);                                    // 0 parameters
4: print(a -> a.startsWith("test"));                     // 1 parameter
5: print((String a) -> a.startsWith("test"));            // 1 parameter
6: print((a, b) -> a.startsWith("test"));                // 2 parameters
7: print((String a, String b) -> a.startsWith("test"));  // 2 parameters
```

Notice that all of these examples have parentheses around the parameter list except the one that takes only one parameter and doesn't specify the type. Line 3 takes 0 parameters and always returns the Boolean `true`. Line 4 takes one parameter and calls a method on it, returning the result. Line 5 does the same except that it explicitly defines the type of the variable. Lines 6 and 7 take two parameters and ignore one of them—there isn't a rule that says you must use all defined parameters.

Now let's make sure you can identify invalid syntax. Do you see what's wrong with each of these?

```
print(a, b -> a.startsWith("test"));                    // DOES NOT COMPILE
print(a -> { a.startsWith("test"); });                  // DOES NOT COMPILE
print(a -> { return a.startsWith("test") });            // DOES NOT COMPILE
```

The first line needs parentheses around the parameter list. Remember that the parentheses are *only* optional when there is one parameter and it doesn't have a type declared. The second line is missing the `return` keyword. The last line is missing the semicolon.

You might have noticed all of our lambdas return a `boolean`. That is because the scope for the OCA exam limits what you need to learn.

Real World Scenario

What Variables Can My Lambda Access?

Lambdas are allowed to access variables. This topic isn't on the OCA exam, but you may come across it when practicing. Lambdas are allowed to access variables. Here's an example:

```
boolean wantWhetherCanHop = true;
print(animals, a -> a.canHop() == wantWhetherCanHop);
```

The trick is that they cannot access all variables. Instance and static variables are okay. Method parameters and local variables are fine if they are not assigned new values.

There is one more issue you might see with lambdas. We've been defining an argument list in our lambda expressions. Since Java doesn't allow us to redeclare a local variable, the following is an issue:

```
(a, b) -> { int a = 0; return 5;}     // DOES NOT COMPILE
```

We tried to redeclare `a`, which is not allowed. By contrast, the following line is okay because it uses a different variable name:

```
(a, b) -> { int c = 0; return 5;}
```

Predicates

In our earlier example, we created an interface with one method:

```
boolean test(Animal a);
```

Lambdas work with interfaces that have only one method. These are called functional interfaces—interfaces that can be used with functional programming. (It's actually more complicated than this, but for the OCA exam this definition is fine.)

You can imagine that we'd have to create lots of interfaces like this to use lambdas. We want to test `Animals` and `Strings` and `Plants` and anything else that we come across.

Luckily, Java recognizes that this is a common problem and provides such an interface for us. It's in the package `java.util.function` and the gist of it is as follows:

```
public interface Predicate<T> {
  boolean test(T t);
}
```

That looks a lot like our method. The only difference is that it uses this type `T` instead of `Animal`. That's the syntax for generics. It's like when we created an `ArrayList` and got to specify any type that goes in it.

This means we don't need our own interface anymore and can put everything related to our search in one class:

```
1: import java.util.*;
2: import java.util.function.*;
3: public class PredicateSearch {
4:   public static void main(String[] args) {
5:     List<Animal> animals = new ArrayList<Animal>();
6:     animals.add(new Animal("fish", false, true));
7:
8:     print(animals, a -> a.canHop());
9:   }
10:  private static void print(List<Animal> animals, Predicate<Animal>↵
     checker) {
11:    for (Animal animal : animals) {
12:      if (checker.test(animal))
13:         System.out.print(animal + " ");
14:     }
15:     System.out.println();
16:  }
17: }
```

This time, line 10 is the only one that changed. We expect to have a `Predicate` passed in that uses type `Animal`. Pretty cool. We can just use it without having to write extra code.

Java 8 even integrated the `Predicate` interface into some existing classes. There is only one you need to know for the exam. `ArrayList` declares a `removeIf()` method that takes a `Predicate`. Imagine we have a list of names for pet bunnies. We decide we want to remove all of the bunny names that don't begin with the letter h because our little cousin really wants us to choose an H name. We could solve this problem by writing a loop. Or we could solve it in one line:

```
3: List<String> bunnies = new ArrayList<>();
4: bunnies.add("long ear");
5: bunnies.add("floppy");
6: bunnies.add("hoppy");
7: System.out.println(bunnies);     // [long ear, floppy, hoppy]
8: bunnies.removeIf(s -> s.charAt(0) != 'h');
9: System.out.println(bunnies);     // [hoppy]
```

Line 8 takes care of everything for us. It defines a predicate that takes a `String` and returns a `boolean`. The `removeIf()` method does the rest.

For the OCA exam, you only need to know how to implement lambda expressions that use the `Predicate` interface. Remember the one method in the interface called `test()`? It takes any one reference type parameter and returns a `boolean`. Functional programming is a large topic and just the basics are covered. On the OCP exam, you'll learn how to get rid of the loop entirely for more than just `removeIf()`. You'll also learn the rules for implementing your own functional interfaces as we did with `CheckTrait`.

Summary

As you learned in this chapter, Java methods start with an access modifier of `public`, `private`, `protected` or blank (default access). This is followed by an optional specifier such as `static`, `final`, or `abstract`. Next comes the return type, which is `void` or a Java type. The method name follows, using standard Java identifier rules. Zero or more parameters go in parentheses as the parameter list. Next come any optional exception types. Finally, zero or more statements go in braces to make up the method body.

Using the `private` keyword means the code is only available from within the same class. Default (package private) access means the code is only available from within the same package. Using the `protected` keyword means the code is available from the same package or subclasses. Using the `public` keyword means the code is available from anywhere. Static methods and static variables are shared by the class. When referenced from outside the class, they are called using the classname—for example, `StaticClass.method()`. Instance members are allowed to call static members, but static members are not allowed to call instance members. Static imports are used to import static members.

Java uses pass-by-value, which means that calls to methods create a copy of the parameters. Assigning new values to those parameters in the method doesn't affect the caller's variables.

Calling methods on objects that are method parameters changes the state of those objects and is reflected in the caller.

Overloaded methods are methods with the same name but a different parameter list. Java calls the most specific method it can find. Exact matches are preferred, followed by wider primitives. After that comes autoboxing and finally varargs.

Constructors are used to instantiate new objects. The default no-argument constructor is called when no constructor is coded. Multiple constructors are allowed and can call each other by writing `this()`. If `this()` is present, it must be the first statement in the constructor. Constructors can refer to instance variables by writing `this` before a variable name to indicate they want the instance variable and not the method parameter with that name. The order of initialization is the superclass (which we will cover in Chapter 5); static variables and static initializers in the order they appear; instance variables and instance initializers in the order they appear; and finally the constructor.

Encapsulation refers to preventing callers from changing the instance variables directly. This is done by making instance variables private and getters/setters public. Immutability refers to preventing callers from changing the instance variables at all. This uses several techniques, including removing setters. JavaBeans use methods beginning with `is` and `get` for `boolean` and non-`boolean` property types, respectively. Methods beginning with `set` are used for setters.

Lambda expressions, or lambdas, allow passing around blocks of code. The full syntax looks like `(String a, String b) -> { return a.equals(b); }`. The parameter types can be omitted. When only one parameter is specified without a type, the parentheses can also be omitted. The braces and return statement can be omitted for a single statement, making the short form `(a -> a.equals(b))`. Lambdas are passed to a method expecting an interface with one method. `Predicate` is a common interface. It has one method named `test` that returns a `boolean` and takes any type. The `removeIf()` method on `ArrayList` takes a `Predicate`.

Exam Essentials

Be able to identify correct and incorrect method declarations. A sample method signature is `public static void method(String... args) throws Exception {}`.

Identify when a method or field is accessible. Recognize when a method or field is accessed when the access modifier (`private`, `protected`, `public`, or default access) does not allow it.

Recognize valid and invalid uses of static imports. Static imports import static members. They are written as `import static`, not *static import*. Make sure they are importing static methods or variables rather than classnames.

State the output of code involving methods. Identify when to call static rather than instance methods based on whether the classname or object comes before the method.

Recognize the correct overloaded method. Exact matches are used first, followed by wider primitives, followed by autoboxing, followed by varargs. Assigning new values to method parameters does not change the caller, but calling methods on them does.

Evaluate code involving constructors. Constructors can call other constructors by calling `this()` as the first line of the constructor. Recognize when the default constructor is provided. Remember the order of initialization is the superclass, static variables/initializers, instance variables/initializers, and the constructor.

Be able to recognize when a class is properly encapsulated. Look for `private` instance variables and `public` getters and setters when identifying encapsulation.

Write simple lambda expressions. Look for the presence or absence of optional elements in lambda code. Parameter types are optional. Braces and the `return` keyword are optional when the body is a single statement. Parentheses are optional when only one parameter is specified and the type is implicit. The `Predicate` interface is commonly used with lambdas because it declares a single method called `test()`, which takes one parameter.

Review Questions

1. Which of the following can fill in the blank in this code to make it compile? (Choose all that apply)

```
public class Ant {
  _____ void method() { }
}
```

   A. `default`
   B. `final`
   C. `private`
   D. `Public`
   E. `String`
   F. `zzz:`

2. Which of the following compile? (Choose all that apply)
   A. `final static void method4() { }`
   B. `public final int void method() { }`
   C. `private void int method() { }`
   D. `static final void method3() { }`
   E. `void final method() {}`
   F. `void public method() { }`

3. Which of the following methods compile? (Choose all that apply)
   A. `public void methodA() {  return;}`
   B. `public void methodB() {  return null;}`
   C. `public void methodD() {}`
   D. `public int methodD() {  return 9;}`
   E. `public int methodE() {  return 9.0;}`
   F. `public int methodF() {  return;}`
   G. `public int methodG() {  return null;}`

4. Which of the following compile? (Choose all that apply)
   A. `public void moreA(int... nums) {}`
   B. `public void moreB(String values, int... nums) {}`
   C. `public void moreC(int... nums, String values) {}`
   D. `public void moreD(String... values, int... nums) {}`
   E. `public void moreE(String[] values, ...int nums) {}`
   F. `public void moreF(String... values, int[] nums) {}`
   G. `public void moreG(String[] values, int[] nums) {}`

5. Given the following method, which of the method calls return 2? (Choose all that apply)

```
public int howMany(boolean b, boolean... b2) {
  return b2.length;
}
```

   A. `howMany();`
   B. `howMany(true);`
   C. `howMany(true, true);`
   D. `howMany(true, true, true);`
   E. `howMany(true, {true});`
   F. `howMany(true, {true, true});`
   G. `howMany(true, new boolean[2]);`

6. Which of the following are true? (Choose all that apply)
   A. Package private access is more lenient than protected access.
   B. A `public` class that has private fields and package private methods is not visible to classes outside the package.
   C. You can use access modifiers so only some of the classes in a package see a particular package private class.
   D. You can use access modifiers to allow read access to all methods, but not any instance variables.
   E. You can use access modifiers to restrict read access to all classes that begin with the word `Test`.

7. Given the following `my.school.ClassRoom` and `my.city.School` class definitions, which line numbers in `main()` generate a compiler error? (Choose all that apply)

```
1: package my.school;
2: public class Classroom {
3:   private int roomNumber;
4:   protected String teacherName;
5:   static int globalKey = 54321;
6:   public int floor = 3;
7:   Classroom(int r, String t) {
8:     roomNumber = r;
9:     teacherName = t; } }

1: package my.city;
2: import my.school.*;
3: public class School {
4:   public static void main(String[] args) {
5:     System.out.println(Classroom.globalKey);
6:     Classroom room = new Classroom(101, ""Mrs. Anderson");
```

```
7:     System.out.println(room.roomNumber);
8:     System.out.println(room.floor);
9:     System.out.println(room.teacherName); } }
```

A. None, the code compiles fine.

B. Line 5

C. Line 6

D. Line 7

E. Line 8

F. Line 9

8. Which of the following are true? (Choose all that apply)

A. Encapsulation uses package private instance variables.

B. Encapsulation uses private instance variables.

C. Encapsulation allows setters.

D. Immutability uses package private instance variables.

E. Immutability uses private instance variables.

F. Immutability allows setters.

9. Which are methods using JavaBeans naming conventions for accessors and mutators? (Choose all that apply)

A. `public boolean getCanSwim() {  return canSwim;}`

B. `public boolean canSwim() {  return numberWings;}`

C. `public int getNumWings() {  return numberWings;}`

D. `public int numWings()    {  return numberWings;}`

E. `public void setCanSwim(boolean b) {  canSwim = b;}`

10. What is the output of the following code?

```
1: package rope;
2: public class Rope {
3:  public static int LENGTH = 5;
4:  static {
5:    LENGTH = 10;
6:  }
```

```
7:  public static void swing() {
8:    System.out.print("swing ");
9:  }
10: }

1: import rope.*;
2: import static rope.Rope.*;
3: public class Chimp {
4:  public static void main(String[] args) {
5:    Rope.swing();
6:    new Rope().swing();
7:    System.out.println(LENGTH);
8:  }
9: }
```

A. `swing swing 5`

B. `swing swing 10`

C. Compiler error on line 2 of `Chimp`.

D. Compiler error on line 5 of `Chimp`.

E. Compiler error on line 6 of `Chimp`.

F. Compiler error on line 7 of `Chimp`.

11. Which are true of the following code? (Choose all that apply)

```
1:  public class Rope {
2:    public static void swing() {
3:      System.out.print("swing ");
4:    }
5:    public void climb() {
6:      System.out.println("climb ");
7:    }
8:    public static void play() {
9:      swing();
10:     climb();
11:   }
12:   public static void main(String[] args) {
13:     Rope rope = new Rope();
14:     rope.play();
15:     Rope rope2 = null;
16:     rope2.play();
17:   }
18: }
```

A. The code compiles as is.

B. There is exactly one compiler error in the code.

C. There are exactly two compiler errors in the code.

D. If the lines with compiler errors are removed, the output is `climb climb`.

E. If the lines with compiler errors are removed, the output is `swing swing`.

F. If the lines with compile errors are removed, the code throws a `NullPointerException`.

12. What is the output of the following code?

```
import rope.*;
import static rope.Rope.*;
public class RopeSwing {
  private static Rope rope1 = new Rope();
  private static Rope rope2 = new Rope();
  {
    System.out.println(rope1.length);
  }
  public static void main(String[] args) {
    rope1.length = 2;
    rope2.length = 8;
    System.out.println(rope1.length);
  }
}

package rope;
public class Rope {
  public static int length = 0;
}
```

A. `02`

B. `08`

C. `2`

D. `8`

E. The code does not compile.

F. An exception is thrown.

13. How many compiler errors are in the following code?

```
1: public class RopeSwing {
2:   private static final String leftRope;
3:   private static final String rightRope;
4:   private static final String bench;
5:   private static final String name = "name";
```

```
6:   static {
7:     leftRope = "left";
8:     rightRope = "right";
9:   }
10:   static {
11:     name = "name";
12:     rightRope = "right";
13:   }
14:   public static void main(String[] args) {
15:     bench = "bench";
16:   }
17: }
```

A. 0

B. 1

C. 2

D. 3

E. 4

F. 5

14. Which of the following can replace line 2 to make this code compile? (Choose all that apply)

```
1: import java.util.*;
2: // INSERT CODE HERE
3: public class Imports {
4:  public void method(ArrayList<String> list) {
5:    sort(list);
6:  }
7: }
```

A. `import static java.util.Collections;`

B. `import static java.util.Collections.*;`

C. `import static java.util.Collections.sort(ArrayList<String>);`

D. `static import java.util.Collections;`

E. `static import java.util.Collections.*;`

F. `static import java.util.Collections.sort(ArrayList<String>);`

15. What is the result of the following statements?

```
1:  public class Test {
2:    public void print(byte x) {
3:      System.out.print("byte");
4:    }
5:    public void print(int x) {
6:      System.out.print("int");
```

```
7:    }
8:    public void print(float x) {
9:      System.out.print("float");
10:   }
11:   public void print(Object x) {
12:     System.out.print("Object");
13:   }
14:   public static void main(String[] args) {
15:     Test t = new Test();
16:     short s = 123;
17:     t.print(s);
18:     t.print(true);
19:     t.print(6.789);
20:   }
21: }
```

A. `bytefloatObject`

B. `intfloatObject`

C. `byteObjectfloat`

D. `intObjectfloat`

E. `intObjectObject`

F. `byteObjectObject`

16. What is the result of the following program?

```
1: public class Squares {
2:   public static long square(int x) {
3:     long y = x * (long) x;
4:     x = -1;
5:     return y;
6:   }
7:   public static void main(String[] args) {
8:     int value = 9;
9:     long result = square(value);
10:     System.out.println(value);
11:   } }
```

A. `-1`

B. `9`

C. `81`

D. Compiler error on line 9.

E. Compiler error on a different line.

17. Which of the following are output by the following code? (Choose all that apply)

```
public class StringBuilders {
  public static StringBuilder work(StringBuilder a,
StringBuilder b) {
    a = new StringBuilder("a");
    b.append("b");
    return a;
  }
  public static void main(String[] args) {
    StringBuilder s1 = new StringBuilder("s1");
    StringBuilder s2 = new StringBuilder("s2");
    StringBuilder s3 = work(s1, s2);
    System.out.println("s1 = " + s1);
    System.out.println("s2 = " + s2);
    System.out.println("s3 = " + s3);
  }
}
```

A. `s1 = a`

B. `s1 = s1`

C. `s2 = s2`

D. `s2 = s2b`

E. `s3 = a`

F. `s3 = null`

G. The code does not compile.

18. Which of the following are true? (Choose 2)

A. `this()` can be called from anywhere in a constructor.

B. `this()` can be called from any instance method in the class.

C. `this.variableName` can be called from any instance method in the class.

D. `this.variableName` can be called from any static method in the class.

E. You must include a default constructor in the code if the compiler does not include one.

F. You can call the default constructor written by the compiler using `this()`.

G. You can access a private constructor with the `main()` method.

19. Which of these classes compile and use a default constructor? (Choose all that apply)

A. `public class Bird { }`

B. `public class Bird { public bird() {} }`

C. `public class Bird { public bird(String name) {} }`

D. `public class Bird { public Bird() {} }`

E. `public class Bird { Bird(String name) {} }`

F. `public class Bird { private Bird(int age) {} }`

G. `public class Bird { void Bird() { } }`

20. Which code can be inserted to have the code print `2`?

```
public class BirdSeed {
  private int numberBags;
  boolean call;

  public BirdSeed() {
    // LINE 1
    call = false;
    // LINE 2
  }
  public BirdSeed(int numberBags) {
    this.numberBags = numberBags;
  }
  public static void main(String[] args) {
    BirdSeed seed = new BirdSeed();
    System.out.println(seed.numberBags);
  } }
```

A. Replace line 1 with `BirdSeed(2);`

B. Replace line 2 with `BirdSeed(2);`

C. Replace line 1 with `new BirdSeed(2);`

D. Replace line 2 with `new BirdSeed(2);`

E. Replace line 1 with `this(2);`

F. Replace line 2 with `this(2);`

21. Which of the following complete the constructor so that this code prints out `50`? (Choose all that apply)

```
public class Cheetah {
  int numSpots;
  public Cheetah(int numSpots) {
    // INSERT CODE HERE
  }
  public static void main(String[] args) {
    System.out.println(new Cheetah(50).numSpots);
  }
}
```

A. `numSpots = numSpots;`

B. `numSpots = this.numSpots;`

C. `this.numSpots = numSpots;`

D. `numSpots = super.numSpots;`

E. `super.numSpots = numSpots;`

F. None of the above.

22. What is the result of the following?

```
1:  public class Order {
2:    static String result = "";
3:    { result += "c"; }
4:    static
5:    { result += "u"; }
6:    { result += "r"; }
7: }

1: public class OrderDriver {
2:   public static void main(String[] args) {
3:     System.out.print(Order.result + " ");
4:     System.out.print(Order.result + " ");
5:     new Order();
6:     new Order();
7:     System.out.print(Order.result + " ");
8:   }
9: }
```

A. `curur`

B. `ucrcr`

C. `u ucrcr`

D. `u u curcur`

E. `u u ucrcr`

F. `ur ur urc`

G. The code does not compile.

23. What is the result of the following?

```
1: public class Order {
2:   String value = "t";
3:   { value += "a"; }
4:   { value += "c"; }
5:   public Order() {
```

```
6:     value += "b";
7:   }
8:   public Order(String s) {
9:     value += s;
10:  }
11:  public static void main(String[] args) {
12:    Order order = new Order("f");
13:    order = new Order();
14:    System.out.println(order.value);
15:  } }
```

A. tacb

B. tacf

C. tacbf

D. tacfb

E. tacftacb

F. The code does not compile.

G. An exception is thrown.

24. Which of the following will compile when inserted in the following code? (Choose all that apply)

```
public class Order3 {
  final String value1 = "1";
  static String value2 = "2";
  String value3 = "3";
  {
    // CODE SNIPPET 1
  }
  static {
    // CODE SNIPPET 2
  }
}
```

A. `value1 = "d";` instead of `// CODE SNIPPET 1`

B. `value2 = "e";` instead of `// CODE SNIPPET 1`

C. `value3 = "f";` instead of `// CODE SNIPPET 1`

D. `value1 = "g";` instead of `// CODE SNIPPET 2`

E. `value2 = "h";` instead of `// CODE SNIPPET 2`

F. `value3 = "i";` instead of `// CODE SNIPPET 2`

25. Which of the following are true about the following code? (Choose all that apply)

```
public class Create {
  Create() {
    System.out.print("1 ");
  }
  Create(int num) {
    System.out.print("2 ");
  }
  Create(Integer num) {
    System.out.print("3 ");
  }
  Create(Object num) {
    System.out.print("4 ");
  }
  Create(int... nums) {
    System.out.print("5 ");
  }
  public static void main(String[] args) {
    new Create(100);
    new Create(1000L);
  }
}
```

A. The code prints out `2 4`.

B. The code prints out `3 4`.

C. The code prints out `4 2`.

D. The code prints out `4 4`.

E. The code prints `3 4` if you remove the constructor `Create(int num)`.

F. The code prints `4 4` if you remove the constructor `Create(int num)`.

G. The code prints `5 4` if you remove the constructor `Create(int num)`.

26. What is the result of the following class?

```
1: import java.util.function.*;
2:
3: public class Panda {
4:   int age;
5:   public static void main(String[] args) {
6:     Panda p1 = new Panda();
7:     p1.age = 1;
8:     check(p1, p -> p.age < 5);
```

```
9:   }
10:   private static void check(Panda panda, Predicate<Panda> pred) {
11:     String result = pred.test(panda) ? "match" : "not match";
12:     System.out.print(result);
13: } }
```

A. `match`

B. `not match`

C. Compiler error on line 8.

D. Compiler error on line 10.

E. Compiler error on line 11.

F. A runtime exception is thrown.

27. What is the result of the following code?

```
1: interface Climb {
2:   boolean isTooHigh(int height, int limit);
3: }
4:
5: public class Climber {
6:   public static void main(String[] args) {
7:   check((h, l) -> h.append(l).isEmpty(), 5);
8:   }
9:   private static void check(Climb climb, int height) {
10:    if (climb.isTooHigh(height, 10))
11:      System.out.println("too high");
12:    else
13:      System.out.println("ok");
14:  }
15: }
```

A. `ok`

B. `too high`

C. Compiler error on line 7.

D. Compiler error on line 10.

E. Compiler error on a different line.

F. A runtime exception is thrown.

28. Which of the following lambda expressions can fill in the blank? (Choose all that apply)

```
List<String> list = new ArrayList<>();
list.removeIf(____________________);
```

A. `s -> s.isEmpty()`

B. `s -> {s.isEmpty()}`

C. `s -> {s.isEmpty();}`

D. `s -> {return s.isEmpty();}`

E. `String s -> s.isEmpty()`

F. `(String s) -> s.isEmpty()`

29. Which lambda can replace the `MySecret` class to return the same value? (Choose all that apply)

```
interface Secret {
  String magic(double d);
}

class MySecret implements Secret {
  public String magic(double d) {
    return "Poof";
  }
}
```

A. `caller((e) -> "Poof");`

B. `caller((e) -> {"Poof"});`

C. `caller((e) -> { String e = ""; "Poof" });`

D. `caller((e) -> { String e = ""; return "Poof"; });`

E. `caller((e) -> { String e = ""; return "Poof" });`

F. `caller((e) -> { String f = ""; return "Poof"; });`